8 Innovations with Linux core files for advanced process forensics

by Ryan O'Neill,
who also publishes as Elfmaster

8.1 Introduction

It has been some time since I've seen any really innovative steps forward in process memory forensics. It remains a somewhat arcane topic, and is understood neither widely nor in great depth. In this article I will try to remedy that, and will assume that the readers already have some background knowledge of Linux process memory forensics and the ELF format.

Many of us have been frustrated by the near-uselessness of Linux (ELF) core files for forensics analysis. Indeed, these files are only useful for debugging, and only if you also have the original executable that the core file was dumped from during crash time. There are some exceptions such as `/proc/kcore` for kernel forensics, but even `/proc/kcore` could use a face-lift. Here I present *ECFS*, a technology I have designed to remedy these drawbacks.

8.2 Synopsis

ECFS (Extended core file snapshots) is a custom Linux core dump handler and snapshot utility. It can be used to plug directly into the core dump handler by using the IPC functionality available by passing the pipe '|' symbol in the `/proc/sys/kernel/core_pattern`. ECFS can also be used to take an *ecfs-snapshot* of a process without killing the process, as is often desirable in automated forensics analysis for whole-system process scanning. In this paper, I showcase ECFS in a series of examples as a means of demonstrating its capabilities. I hope to convince you how useful these capabilities will be in modern forensics analysis of Linux process images—which should speak to all forms of binary and process-memory malware analysis. My hope is that ECFS will help revolutionize automated detection of process memory anomalies.

ECFS creates files that are backward-compatible with regular core files but are also prolific in new features, including section headers (which core files do not have) and many *new* section headers and section header types. ECFS includes full symbol table reconstruction for both `.dynsym` and `.symtab` symbol tables. Regular core files do not have section headers or symbol tables (and rely on having the original executable for such things), whereas an *ecfs-core* contains everything a forensics analyst would ever want, in one package.

Since the object and `readelf` output of an *ecfs-core* file is huge, let us examine a simple *ecfs-core* for a 64-bit ELF program named `host`. The process for `host` will show some signs of virus memory infection or backdooring, which ECFS will help bring to light.

The following command will set up the kernel core handler so that it pipes core files into the *stdin* of our core–to–ecfs conversion program named `ecfs`.

```
# echo '|/opt/ecfs/bin/ecfs -i -e %e -p %p -o /opt/ecfs/cores/%e.%p' > /proc/sys/kernel/
    core_pattern
```

Next, let's get the kernel to dump an *ecfs* file of the process for `host`, and then begin analyzing this file.

```
$ kill -11 `pidof host`
```

8.3 Section header reconstruction example

```
$ readelf -S /opt/ecfs/cores/host.10710
```

There are 40 section headers, starting at offset `0x23fff0`:

```
Section Headers:
  [Nr] Name              Type             Address           Offset
       Size              EntSize          Flags  Link  Info  Align
  [ 0]                   NULL             0000000000000000  00000000
       0000000000000000  0000000000000000           0     0     0
  [ 1] .interp           PROGBITS         0000000000400238  00002238
       000000000000001c  0000000000000000   A       0     0     1
  [ 2] .note             NOTE             0000000000000000  000004a0
       0000000000000bd8  0000000000000000   A       0     0     4
  [ 3] .hash             GNU_HASH         0000000000400298  00002298
       000000000000001c  0000000000000000   A       0     0     4
  [ 4] .dynsym           DYNSYM           00000000004002b8  000022b8
       00000000000000a8  0000000000000018   A       5     0     8
  [ 5] .dynstr           STRTAB           0000000000400360  00002360
       0000000000000050  0000000000000018   A       0     0     1
  [ 6] .rela.dyn         RELA             00000000004003e0  000023e0
       0000000000000018  0000000000000018   A       4     0     8
  [ 7] .rela.plt         RELA             00000000004003f8  000023f8
       0000000000000090  0000000000000018   A       4     0     8
  [ 8] .init             PROGBITS         0000000000400488  00002488
       000000000000001a  0000000000000000  AX       0     0     8
  [ 9] .plt              PROGBITS         00000000004004b0  000024b0
       0000000000000070  0000000000000010  AX       0     0     16
  [10] .text             PROGBITS         0000000000400000  00002000
       0000000000001000  0000000000000000  AX       0     0     16
  [11] .fini             PROGBITS         0000000000400724  00002724
       0000000000000009  0000000000000000  AX       0     0     16
  [12] .eh_frame_hdr     PROGBITS         0000000000400758  00002758
       0000000000000034  0000000000000000  AX       0     0     4
  [13] .eh_frame         PROGBITS         000000000040078c  00002790
       00000000000000f4  0000000000000000  AX       0     0     8
  [14] .dynamic          DYNAMIC          0000000000600e28  00003e28
       00000000000001d0  0000000000000010  WA       0     0     8
  [15] .got.plt          PROGBITS         0000000000601000  00004000
       0000000000000050  0000000000000008  WA       0     0     8
  [16] .data             PROGBITS         0000000000600000  00003000
       0000000000001000  0000000000000000  WA       0     0     8
  [17] .bss              PROGBITS         0000000000601058  00004058
       0000000000000008  0000000000000000  WA       0     0     8
  [18] .heap             PROGBITS         000000000093b000  00005000
       0000000000021000  0000000000000000  WA       0     0     8
  [19] ld-2.19.so.text   SHLIB            0000003000000000  00026000
       0000000000023000  0000000000000000   A       0     0     8
  [20] ld-2.19.so.relro  SHLIB            0000003000222000  00049000
       0000000000001000  0000000000000000   A       0     0     8
  [21] ld-2.19.so.data.0 SHLIB            0000003000223000  0004a000
       0000000000001000  0000000000000000   A       0     0     8
  [22] libc-2.19.so.text SHLIB            0000003001000000  0004c000
       00000000001bb000  0000000000000000   A       0     0     8
  [23] libc-2.19.so.unde SHLIB            00000030011bb000  00207000
       0000000000200000  0000000000000000   A       0     0     8
  [24] libc-2.19.so.relr SHLIB            00000030013bb000  00207000
       0000000000004000  0000000000000000   A       0     0     8
  [25] libc-2.19.so.data SHLIB            00000030013bf000  0020b000
       0000000000002000  0000000000000000   A       0     0     8
  [26] evil_lib.so.text  INJECTED         00007fb0358c3000  00215000
       0000000000002000  0000000000000000   A       0     0     8
  [27] .prstatus         PROGBITS         0000000000000000  0023f000
       0000000000000150  0000000000000150           0     0     4
  [28] .fdinfo           PROGBITS         0000000000000000  0023f150
       0000000000000c78  0000000000000214           0     0     4
  [29] .siginfo          PROGBITS         0000000000000000  0023fdc8
       0000000000000080  0000000000000080           0     0     4
  [30] .auxvector        PROGBITS         0000000000000000  0023fe48
       0000000000000130  0000000000000008           0     0     8
  [31] .exepath          PROGBITS         0000000000000000  0023ff78
       0000000000000024  0000000000000008           0     0     1
  [32] .personality      PROGBITS         0000000000000000  0023ff9c
       0000000000000004  0000000000000004           0     0     1
  [33] .arglist          PROGBITS         0000000000000000  0023ffa0
       0000000000000050  0000000000000001           0     0     1
  [34] .stack            PROGBITS         00007fff51d82000  00000000
       0000000000021000  0000000000000000  WA       0     0     8
  [35] .vdso             PROGBITS         00007fff51dfe000  0023c000
       0000000000002000  0000000000000000  WA       0     0     8
```

```
  [36] .vsyscall        PROGBITS         ffffffffff600000  0023e000
       0000000000001000 0000000000000000  WA       0     0     8
  [37] .symtab          SYMTAB           0000000000000000  00240b81
       0000000000000078 0000000000000018          38     0     4
  [38] .strtab          STRTAB           0000000000000000  00240bf9
       0000000000000037 0000000000000000           0     0     1
  [39] .shstrtab        STRTAB           0000000000000000  002409f0
       0000000000000191 0000000000000000           0     0     1
```

As you can see, there are even more section headers in our *ecfs-core* file than in the original executable itself. This means that you can disassemble a complete process image with simple tools that rely on section headers such as `objdump`! Also, please note this file is entirely usable as a regular core file; the only change you must make to it is to mark it from `ET_NONE` to `ET_CORE` in the initial ELF file header. The reason it is marked as `ET_NONE` is that `objdump` would know to utilize the section headers instead of the program headers.

```
$ tools/et_flip host.107170  <- this command flips e_type from ET_NONE to ET_CORE (And vice versa)
$ gdb -q host host.107170
[New LWP 10710]
Core was generated by `ecfs_tests/host'.
Program terminated with signal SIGSEGV, Segmentation fault.
#0  0x00007fb0358c375a in ?? ()
(gdb) bt
#0  0x00007fb0358c375a in ?? ()
#1  0x00007fff51da1580 in ?? ()
#2  0x00007fb0358c3790 in ?? ()
#3  0x0000000000000000 in ?? ()
```

For the remainder of this paper we will not be using traditional core file functionality. However, it is important to know that it's still available.

So what new sections do we see that have never existed in traditional ELF files? Well, we have sections for important memory segments from the process that can be navigated by name with section headers. Much easier than having to figure out which program header corresponds to which mapping!

```
  [18] .heap            PROGBITS         000000000093b000  00005000
       0000000000021000 0000000000000000  WA       0     0     8
  [34] .stack           PROGBITS         00007fff51d82000  00000000
       0000000000021000 0000000000000000  WA       0     0     8
  [35] .vdso            PROGBITS         00007fff51dfe000  0023c000
       0000000000002000 0000000000000000  WA       0     0     8
  [36] .vsyscall        PROGBITS         ffffffffff600000  0023e000
       0000000000001000 0000000000000000  WA       0     0     8
```

Also notice that there are section headers for every mapping of each shared library. For instance, the dynamic linker is mapped in as it usually is:

```
  [19] ld-2.19.so.text  SHLIB            0000003000000000  00026000
       0000000000023000 0000000000000000   A       0     0     8
  [20] ld-2.19.so.relro SHLIB            0000003000222000  00049000
       0000000000001000 0000000000000000   A       0     0     8
  [21] ld-2.19.so.data.0 SHLIB           0000003000223000  0004a000
       0000000000001000 0000000000000000   A       0     0     8
```

Also notice the section type is `SHLIB`. This was a reserved type specified in the ELF man pages that is never used, so I thought this to be the perfect opportunity for it to see some action. Notice how each part of the shared library is given its own section header: `<lib>.text` for the code segment, `<lib>.relro` for the read-only page to help protect against `.got.plt` and `.dtors` overwrites, and `<lib>.data` for the data segment.

Another important thing to note is that in traditional core files only the first 4,096 bytes of the main executable and each shared libraries' text images are written to disk. This is done to save space, and, considering that the text segment presumably should not change, this is usually OK. However, in forensics analysis we must be open to the possibility of an `RWX` text segment that has been modified, e.g., with inline function hooking.

8.4 Heuristics

Also notice that there is one section showing a suspicious-looking shared library that is not marked as the type `SHLIB` but instead as `INJECTED`.

```
  [26] evil_lib.so.text   INJECTED          00007fb0358c3000   00215000
       0000000000002000   0000000000000000   A        0     0     8
```

"`#define SHT_INJECTED 0x200000`" is custom and the `readelf` utility has been modified on my system to reflect this. A standard `readelf` will show it as `<unknown>`.

This section is for a shared library that was considered by *ecfs* to be maliciously injected into the process. The *ecfs* core handler does quite a bit of heuristics work on its own, and therefore leaves very little work for the forensic analyst. In other words, the analyst no longer needs to know jack about ELF in order to detect complex memory infections (more on this with the PLT/GOT hook detection later!)

Note that these heuristics are enabled by passing the `-h` switch to `/opt/bin/ecfs`. Currently, there are occasional false-positives, and for people designing their own heuristics it might be useful to turn the ecfs-heuristics off.

8.5 Custom section headers

Moving on, there are a number of other custom sections that bring to light a lot of information about the process.

```
  [27] .prstatus          PROGBITS          0000000000000000   0023f000
       0000000000000150   0000000000000150        0     0     4
  [28] .fdinfo            PROGBITS          0000000000000000   0023f150
       0000000000000c78   0000000000000214        0     0     4
  [29] .siginfo           PROGBITS          0000000000000000   0023fdc8
       0000000000000080   0000000000000080        0     0     4
  [30] .auxvector         PROGBITS          0000000000000000   0023fe48
       0000000000000130   0000000000000008        0     0     8
  [31] .exepath           PROGBITS          0000000000000000   0023ff78
       0000000000000024   0000000000000008        0     0     1
  [32] .personality       PROGBITS          0000000000000000   0023ff9c
       0000000000000004   0000000000000004        0     0     1
  [33] .arglist           PROGBITS          0000000000000000   0023ffa0
       0000000000000050   0000000000000001        0     0     1
```

I will not go into complete detail for all of these, but will later show you a simple parser I wrote using the `libecfs` API that is designed specifically to parse *ecfs-core* files. You can probably guess as to what most of these contain, as they are somewhat straightforward; i.e., `.auxvector` contains the process' auxiliary vector, and `.fdinfo` contains data about the file descriptors, sockets, and pipes within the process, including TCP and UDP network information. Finally, `.prstatus` contains `elf_prstatus` and similar structs.

8.6 Symbol table resolution

One of the most powerful features of *ecfs* is the ability to reconstruct full symbol tables for all functions.

```
$ readelf -s host.10710

Symbol table '.dynsym' contains 7 entries:
```

```
    Num:    Value          Size Type    Bind   Vis      Ndx Name
      0: 0000000000000000     0 NOTYPE  LOCAL  DEFAULT  UND
      1: 000000300106f2c0     0 FUNC    GLOBAL DEFAULT  UND fputs
      2: 0000003001021dd0     0 FUNC    GLOBAL DEFAULT  UND __libc_start_main
      3: 000000300106edb0     0 FUNC    GLOBAL DEFAULT  UND fgets
      4: 00007fb0358c3000     0 NOTYPE  WEAK   DEFAULT  UND __gmon_start__
      5: 000000300106f070     0 FUNC    GLOBAL DEFAULT  UND fopen
      6: 00000030010c1890     0 FUNC    GLOBAL DEFAULT  UND sleep

Symbol table '.symtab' contains 5 entries:
    Num:    Value          Size Type    Bind   Vis      Ndx Name
      0: 00000000004004b0   112 FUNC    GLOBAL DEFAULT   10 sub_4004b0
      1: 0000000000400520    42 FUNC    GLOBAL DEFAULT   10 sub_400520
      2: 000000000040060d   160 FUNC    GLOBAL DEFAULT   10 sub_40060d
      3: 00000000004006b0   101 FUNC    GLOBAL DEFAULT   10 sub_4006b0
      4: 0000000000400720     2 FUNC    GLOBAL DEFAULT   10 sub_400720
```

Notice that the dynamic symbols (`.dynsym`) have values that actually reflect the location of where those symbols should be at runtime. If you look at the `.dynsym` of the original executable, you would see those values all zeroed out. With the `.symtab` symbol table, all of the original function locations and sizes have been reconstructed by performing analysis of the exception handling frame descriptors found in the `PT_GNU_EH_FRAME` segment of the program in memory.[37]

8.7 Relocation entries and PLT/GOT hooks

Another very useful feature is the fact that *ecfs-core* files have complete relocation entries, which show the actual runtime relocation values—or rather what you should *expect* this value to be. This is extremely handy for detecting modification of the global offset table found in `.got.plt` section.

```
$ readelf -r host.10710

Relocation section '.rela.dyn' at offset 0x23e0 contains 1 entries:
  Offset          Info           Type           Sym. Value    Sym. Name + Addend
000000600ff8  000400000006 R_X86_64_GLOB_DAT 00007fb0358c3000 __gmon_start__ + 0

Relocation section '.rela.plt' at offset 0x23f8 contains 6 entries:
  Offset          Info           Type           Sym. Value    Sym. Name + Addend
000000601018  000100000007 R_X86_64_JUMP_SLO 000000300106f2c0 fputs + 0
000000601020  000200000007 R_X86_64_JUMP_SLO 0000003001021dd0 __libc_start_main + 0
000000601028  000300000007 R_X86_64_JUMP_SLO 000000300106edb0 fgets + 0
000000601030  000400000007 R_X86_64_JUMP_SLO 00007fb0358c3000 __gmon_start__ + 0
000000601038  000500000007 R_X86_64_JUMP_SLO 000000300106f070 fopen + 0
000000601040  000600000007 R_X86_64_JUMP_SLO 00000030010c1890 sleep + 0
```

Notice that the symbol values for the `.rela.plt` relocation entries actually show what the GOT should be pointing to. For instance:

```
000000601028  000300000007 R_X86_64_JUMP_SLO 000000300106edb0 fgets + 0
```

This means that `0x601028` should be pointing at `0x300106edb0`, unless of course it hasn't been resolved yet, in which case it should point to the appropriate PLT entry. In other words, if `0x601028` has a value that is not `0x300106edb0` and is not the corresponding PLT entry, then you have discovered malicious PLT/GOT hooks in the process. The `libecfs` API comes with a function that makes this heuristic extremely trivial to perform.

[37] I cover this nifty technique in more detail at `http://www.bitlackeys.org/#eh_frame`.

8.8 Libecfs Parsing and Detecting DLL Injection

Still sticking with our `host.10710 ecfs-core` file, let us take a look at the output of `readecfs`, a parsing program I wrote. It's a very small C program; its power comes from using `libecfs`.

```
$ ./readecfs ../infected/host.10710
- read_ecfs output for file ../infected/host.10710
- Executable path (.exepath): /home/ryan/git/ecfs/ecfs_tests/host
- Thread count (.prstatus): 1
- Thread info (.prstatus)
        [thread 1] pid: 10710

- Exited on signal (.siginfo): 11
- files/pipes/sockets (.fdinfo):
        [fd: 0] path: /dev/pts/8
        [fd: 1] path: /dev/pts/8
        [fd: 2] path: /dev/pts/8
        [fd: 3] path: /etc/passwd
        [fd: 4] path: /tmp/passwd_info
        [fd: 5] path: /tmp/evil_lib.so

assigning
- Printing shared library mappings:
ld-2.19.so.text
ld-2.19.so.relro
ld-2.19.so.data.0
libc-2.19.so.text
libc-2.19.so.undef
libc-2.19.so.relro
libc-2.19.so.data.1
evil_lib.so.text // HMM INTERESTING

.dynsym:  - 0
.dynsym: fputs - 300106f2c0
.dynsym: __libc_start_main - 3001021dd0
.dynsym: fgets - 300106edb0 // OF IMPORTANCE
.dynsym: __gmon_start__ - 7fb0358c3000
.dynsym: fopen - 300106f070
.dynsym: sleep - 30010c1890

.symtab: sub_4004b0 - 4004b0
.symtab: sub_400520 - 400520
.symtab: sub_40060d - 40060d
.symtab: sub_4006b0 - 4006b0
.symtab: sub_400720 - 400720

- Printing out GOT/PLT characteristics (pltgot_info_t):
gotsite: 601018 gotvalue: 300106f2c0 gotshlib: 300106f2c0 pltval: 4004c6
gotsite: 601020 gotvalue: 3001021dd0 gotshlib: 3001021dd0 pltval: 4004d6
gotsite: 601028 gotvalue: 7fb0358c3767 gotshlib: 300106edb0 pltval: 4004e6 // WHAT IS WRONG HERE?
gotsite: 601030 gotvalue: 4004f6 gotshlib: 7fb0358c3000 pltval: 4004f6
gotsite: 601038 gotvalue: 300106f070 gotshlib: 300106f070 pltval: 400506
gotsite: 601040 gotvalue: 30010c1890 gotshlib: 30010c1890 pltval: 400516

- Printing auxiliary vector (.auxilliary):
AT_PAGESZ: 1000
AT_PHDR: 400040
AT_PHENT: 38
AT_PHNUM: 9
AT_BASE: 0
AT_FLAGS: 0
AT_ENTRY: 400520
AT_UID: 0
AT_EUID: 0
AT_GID: 0

- Displaying ELF header:
e_entry: 0x400520
e_phnum: 20
e_shnum: 40
e_shoff: 0x23fff0
e_phoff: 0x40
e_shstrndx: 39

--- truncated rest of output ---
```

Just from this output alone, you can see so much about the program that was running, including that at some point a file named `/tmp/evil_lib.so` was opened, and—as we saw from the section header output earlier—it was also mapped into the process.

```
 [26] evil_lib.so.text   INJECTED          00007fb0358c3000   00215000
      0000000000002000   0000000000000000   A        0      0      8
```

Not just mapped in, but injected—as shown by the section header type `SHT_INJECTED`. Another red flag can be seen by examining the line from my parser that I commented on with the note "WHAT IS WRONG HERE?"

```
gotsite: 601028 gotvalue: 7fb0358c3767 gotshlib: 300106edb0 pltval: 4004e6
```

The `gotvalue` is `0x7fb0358c3767`, yet it should be pointing to `0x300106edb0` or `0x4004e6`. Notice anything about the address that it's pointing to? This address `0x7fb0358c3767` is within the range of `evil_lib.so`. As mentioned before it *should* be pointing at `0x300106edb0`, which corresponds to what exactly? Well, let's take a look.

```
$ readelf -r host.10710 | grep 300106edb0
000000601028  000300000007 R_X86_64_JUMP_SLO 000000300106edb0 fgets + 0
```

So we now know that `fgets()` is being hijacked through a PLT/GOT hook! This type of infection has been historically somewhat difficult to detect, so thank goodness that ECFS performed all of the hard work for us.

To further demonstrate the power and ease-of-use that ECFS offers, let us write a very simple memory virus/backdoor forensics scanner that can detect shared library (DLL) injection and PLT/GOT hooking. Writing something like this without `libecfs` would typically take a few thousand lines of C code.

```
-- detect_dll_infection.c --

#include "../libecfs.h"

int main(int argc, char **argv)
{
        ecfs_elf_t *desc;
        ecfs_sym_t *dsyms, *lsyms;
        char *progname;
        int i;
        char *libname;
  ecfs_sym_t *dsyms;
  unsigned long evil_addr;

        if (argc < 2) {
                printf("Usage: %s <ecfs_file>\n", argv[0]);
                exit(0);
        }

        desc = load_ecfs_file(argv[1]);
        progname = get_exe_path(desc);

        for (i = 0; i < desc->ehdr->e_shnum; i++) {
                if (desc->shdr[i].sh_type == SHT_INJECTED) {
                        libname = strdup(&desc->shstrtab[desc->shdr[i].sh_name]);
                        printf("[!] Found maliciously injected shared library: %s\n", libname);
                }
        }
        pltgot_info_t *pltgot;
        int ret = get_pltgot_info(desc, &pltgot);
```

```
        for (i = 0; i < ret; i++) {
                if (pltgot[i].got_entry_va != pltgot[i].shl_entry_va && pltgot[i].got_entry_va !=
    pltgot[i].plt_entry_va)
                        printf("[!] Found PLT/GOT hook, function 'name' is pointing at %lx instead
     of %lx\n",
                                pltgot[i].got_entry_va, evil_addr = pltgot[i].shl_entry_va);
        }
  ret = get_dynamic_symbols(desc, &dsyms);
        for (i = 0; i < ret; i++) {
    if (dsyms[i].symval == evil_addr) {
      printf("[!] %lx corresponds to hijacked function: %s\n", dsyms[i].symval, &dsyms[i].strtab[
    dsyms[i].nameoffset]);
      break;
    }
  }
}
```

This program analyzes an *ecfs-core* file and detects both shared library injection and PLT/GOT hooking used for function hijacking. Let's now run it on our *ecfs* file.

```
$ ./detect_dll_infection host.10710
[!] Found malicously injected shared library: evil_lib.so.text
[!] Found PLT/GOT hook, function 'name' is pointing at 7fb0358c3767 instead of 300106edb0
[!] 300106edb0 corresponds to hijacked function: fgets
```

With just simple forty lines of C code, we have an advanced detection tool capable of detecting an advanced memory infection technique, commonly used by attackers to backdoor a system with a rootkit or virus.

8.9 In Closing

If you liked this paper and are interested in using or contributing to ECFS, feel free to contact me. It will be made available to the public in the near future.[38]

Shouts to Orangetoaster, Baron, Mothra, Dk, Sirus, and Per for ideas, support and feedback regarding this project.

[38] `http://github.com/elfmaster/ecfs`